SMOKE: Scalable Path-Sensitive Memory Leak Detection for Millions of Lines of Code

Gang Fan*, Rongxin Wu*, Qingkai Shi*, Xiao Xiao†, Jinguo Zhou†, Charles Zhang‡
*Hong Kong University of Science and Technology
{gfan, wurongxin, qshiaa, charlesz}@cse.ust.hk
†Sourcebrella Inc.
{xx, jinguo}@sbrella.com

Abstract—Detecting memory leak at industrial scale is still not well addressed, in spite of the tremendous effort from both industry and academia in the past decades. Existing work suffers from an unresolved paradox—a highly precise analysis limits its scalability and an imprecise one seriously hurts its precision or recall. In this work, we present SMOKE, a staged approach to resolve this paradox. In the first stage, instead of using a uniform precise analysis for all paths, we use a scalable but imprecise analysis to compute a succinct set of candidate memory leak paths. In the second stage, we leverage a more precise analysis to verify the feasibility of those candidates. The first stage is scalable, due to the design of a new sparse program representation, the use-flow graph (UFG), that models the problem as a polynomial-time state analysis. The second stage analysis is both precise and efficient, due to the smaller number of candidates and the design of a dedicated constraint solver. Experimental results show that SMOKE can finish checking industrial-sized projects, up to 8MLoC, in forty minutes with an average false positive rate of 24.4%. Besides, SMOKE is significantly faster than the state-of-the-art research techniques as well as the industrial tools, with the speedup ranging from 5.2X to 22.8X. In the twenty-nine mature and extensively checked benchmark projects, SMOKE has discovered thirty previously-unknown memory leaks which were confirmed by developers, and one even assigned a CVE ID.

Index Terms—memory leak, static bug finding, use-flow graph, value-flow graph

I. INTRODUCTION

Despite the tremendous research progress in recent decades [1]–[9], the detection of memory leaks in industrial-scale is still pretty much an unsolved problem. In the first half of the year 2018, more than 680 memory leak bugs have been reported in Firefox [10] and Chrome [11]. More than 240 CVE (Common Vulnerabilities and Exposures) entries in 2017 are memory leaks bugs [12]. Apparently, with the explosive growth of the code size and the complexity in modern software [13], a practical memory detector needs to be highly scalable, checking millions of lines of code within minutes, and precise, understanding complex path conditions with less than 30% false positives [14], [15].

The state-of-the-art approaches suffer from the scalability and precision paradox. One category of the approaches [4]–[9] give up path sensitivity for scalability, inevitably introducing imprecise results. For example, we observed that, SABER [9], a recent path-insensitive memory leak detection technique, incurs a false positive rate of 66.7% in our evaluation. Another category [1]–[3] traverse the control flow graph and use the path-sensitive analysis to achieve high precision. However, they are known to easily suffer from scalability issues, especially for the whole-program analysis. For example, SATURN [3] is reported to have spent more than 23 hours in checking memory leaks for a 5MLoC code base. Our experiment shows that CSA [1] and INFER [2] fail to analyze large projects of over 2MLoC in two hours.

Our idea to resolve this paradox is based on an observation that, in real programs, only a small proportion of program paths lead to memory leaks. Therefore, instead of using a sledge hammer, i.e., the expensive path-sensitive analysis, for all paths, we use a two-staged analysis by first computing a succinct set of candidate memory leak paths through a novel scalable and path-insensitive method, followed by a more precise and heavy-weight verification of the feasibility of these paths, in order to achieve path-sensitivity.

More specifically, to check millions of lines of code in minutes, we believe that the sparse value-flow analysis, already widely adopted in finding memory leaks [8], [9], is the right direction as it tracks values along the data dependence relations on the value flow graph (VFG) instead of the control flow graph, skipping irrelevant program statements to achieve scalability. However, we observe that the VFG, originally intended for program transformations [16], is not suitable for arbitrary finite-state-machine properties such as the memory leak problem, due to the omission of flow information(order of events) [8]. Therefore, instead of finding a leak path, VFG-based methods need to deduce the leak path by checking whether the non-leak paths (i.e., paths where the heap object is safely freed) cover all control flow paths from where the heap memory is allocated. Such an analysis is equivalent to solving a $k-$SAT problem, which is NP-hard with the input size $k > 2$ ($k$ represents the number of the branch conditions) [17]. This induces a very high time complexity in theory and may greatly compromise the efficiency and scalability in practice. For instance, the most recent VFG technique, Pinpoint [18], in spite of its leap in achieving scalability and precision, still cannot complete the analysis in some large subjects in our

§Rongxin Wu is the corresponding author.
To overcome this limitation, we designed an extension of VFG, namely the use-flow graph (UFG), that encodes not just the definition but also the use of the problem-relevant heap objects. All use sites of the same heap object in UFG are ordered according to the control flow, to check finite-state-machine properties and to use polynomial-time graph search methods to find possible memory leak paths. Path-sensitivity is achieved in the second stage by using a dedicated constraint solver to verify the feasibility of these candidate leak paths. The verification process is efficient because, at this stage, the number of paths is very small (only 21 on average) and we use a customized constraint solver to further filter out “easy-to-contradict” ones, leading to a further pruning of the paths. Finally, we invoke a full SMT solver, such as Z3, for the remaining candidates.

To evaluate the scalability and effectiveness of our proposed technique, we implemented a tool, SMOKE, and applied it to the SPEC2000 benchmark programs and seventeen well-maintained open source projects. The experimental results have demonstrated that SMOKE is highly efficient and effective, as it could finish checking industrial-sized projects, up to 8MLoC, to a memory-leak vulnerability. Note that, in general, solving $F_{\text{leak}}$ is a $k$-$\text{SAT}$ problem [17] ($k$ represents the number of the branch conditions), which has the best time complexity of $O((2 - \frac{1}{2^k})^n)$ [19] and is proved to be $\text{NP}$-hard [17] for $k > 2$.

In SMOKE, we propose a new type of sparse program representation, namely use-flow graph (UFG), that still contains the necessary control-flow information. Specifically, the UFG encodes the definition and the use of problem-relevant heap objects, which is similar to the sparseness feature in VFG. In addition, it records the control flow order of uses of the heap objects, which is similar to the sparseness feature in VFG. Meanwhile, UFG explicitly models the necessary data dependence and the control dependence. Meanwhile, UFG explicitly models the life cycle of a value by creating an $\text{out-of-scope}$ node (i.e., $p@s_{88}$ in Figure 1(c)), which indicates that the heap object pointed to by $p$ is no longer referenced. A simple graph traversal on the UFG shall discover that there exits a path where the pointer $p$ is never “freed”. Similarly, a simple graph traversal on the UFG in Figure 2(c) will verify the absence of memory leaks, because each path in the graph contains a memory release operation. It is noteworthy that, using UFG, we can obtain the same results with the same precision as that of FASTCHECK, without collecting constraints for each path and using a heavyweight constraint solver.

B. Regaining Path-Sensitivity via Staged Constraint Solving

Since most of the allocated memory spaces are managed properly in common software, many cases of memory leaks can be detected by graph traversals on the UFG with the flow-sensitive precision. However, there still exists a handful of cases that require capturing branch correlations, the importance of which is illustrated by Figure 3. This example is leak-free because the memory is allocated and released under the dynamic memory allocation and deallocation. To overcome this limitation, we designed an extension of VFG, namely the use-flow graph (UFG), that encodes not just the definition but also the use of the problem-relevant heap objects. All use sites of the same heap object in UFG are ordered according to the control flow, to check finite-state-machine properties and to use polynomial-time graph search methods to find possible memory leak paths. Path-sensitivity is achieved in the second stage by using a dedicated constraint solver to verify the feasibility of these candidate leak paths. The verification process is efficient because, at this stage, the number of paths is very small (only 21 on average) and we use a customized constraint solver to further filter out “easy-to-contradict” ones, leading to a further pruning of the paths. Finally, we invoke a full SMT solver, such as Z3, for the remaining candidates.

To evaluate the scalability and effectiveness of our proposed technique, we implemented a tool, SMOKE, and applied it to the SPEC2000 benchmark programs and seventeen well-maintained open source projects. The experimental results have demonstrated that SMOKE is highly efficient and effective, as it could finish checking industrial-sized projects, up to 8MLoC, to a memory-leak vulnerability. Note that, in general, solving $F_{\text{leak}}$ is a $k$-$\text{SAT}$ problem [17] ($k$ represents the number of the branch conditions), which has the best time complexity of $O((2 - \frac{1}{2^k})^n)$ [19] and is proved to be $\text{NP}$-hard [17] for $k > 2$.

In SMOKE, we propose a new type of sparse program representation, namely use-flow graph (UFG), that still contains the necessary control-flow information. Specifically, the UFG encodes the definition and the use of problem-relevant heap objects, which is similar to the sparseness feature in VFG. In addition, it records the control flow order of uses of the heap objects, which is similar to the sparseness feature in VFG. Meanwhile, UFG explicitly models the necessary data dependence and the control dependence. Meanwhile, UFG explicitly models the life cycle of a value by creating an $\text{out-of-scope}$ node (i.e., $p@s_{88}$ in Figure 1(c)), which indicates that the heap object pointed to by $p$ is no longer referenced. A simple graph traversal on the UFG shall discover that there exits a path where the pointer $p$ is never “freed”. Similarly, a simple graph traversal on the UFG in Figure 2(c) will verify the absence of memory leaks, because each path in the graph contains a memory release operation. It is noteworthy that, using UFG, we can obtain the same results with the same precision as that of FASTCHECK, without collecting constraints for each path and using a heavyweight constraint solver.
constraint-solving methods. for many paths and the high complexity of conventional caused by the large overhead of collecting path conditions the negligence of the branch correlation. Such negligence is same condition at Line 4 and Line 7, respectively. However, the pointer relations in the program have been resolved. We assume that a program consists of functions in SSA form and dedicated constraint solver (Section III-B). In this section, we Second, we achieve the precision of path-sensitivity through a program representation, the use-flow graph (Section III-A). In this section, we use a lightweight finite-state analysis with a new sparse losing the precision, we make two design decisions. First, remaining paths with complex path conditions, we adopt a feasibility to achieve a low false positive rate. To correctly identify that the code snippet is leak-free, SMOKE employs a staged constraint solving process to verify the feasibility of the candidate leak paths, such as the one in Figure 4. First, we adopt a linear-time solver to filter obvious infeasible paths. Most of the false warnings involving branch correlations can be pruned at this stage. For the remaining paths with complex path conditions, we adopt a fully-featured SMT solver, such as Z3 [20], to check their feasibility to achieve a low false positive rate.

III. MEMORY LEAK DETECTION

In order to speed up the memory leak detection without losing the precision, we make two design decisions. First, we use a lightweight finite-state analysis with a new sparse program representation, the use-flow graph (Section III-A). Second, we achieve the precision of path-sensitivity through a dedicated constraint solver (Section III-B). In this section, we assume that a program consists of functions in SSA form and the pointer relations in the program have been resolved. We discuss how to satisfy the assumptions in our implementation in the next section.

A. Finite State Analysis with Use Flow Graph

Our analysis is based on a new type of sparse program representation, the use-flow graph (UFG). Compared to the conventional sparse value-flow graph, UFGs have the following features for checking the finite-state properties of a program:

- It models the whole life cycle of program variables by encoding the property-related control flows, which we referred to as use flows.
- It can be efficiently built in linear time with regard to the program size.

A finite-state property of a program can be modeled as a finite-state machine (FSM), which defines the valid sequences of operations that can be performed upon an object.

Definition 1 (Finite-State Machine (FSM)). A finite-state machine of a program property is a quintuple $M = (\Sigma, S, s_0, \delta, F)$, where

- $\Sigma$ is a finite, non-empty set of classes of program points.
- $S$ is a finite, non-empty set of the states of a program object.
- $s_0 \in S$ is the initial state.
- $\delta$ is the state-transition function: $\delta : S \times \Sigma \rightarrow S$. $\delta((s_1, P)) = s_2$ means that the state $s_1$ of an object can transit to the state $s_2$ if the object goes through a program point $p \in P$.
- $F \subseteq S$ is the set of final states.

Example 1. As shown in Figure 4, we use an FSM to model different states and state transitions of a heap object. There are four states in the FSM: Allocated (A), Freed (F), Error (E), and eXit (X). A is the initial state, and both E and X are the final states. E indicates a safety violation of the finite-state property, such as memory leak and double free, and X denotes a normal exit. The state A will transit to the state F if a newly-allocated heap object goes through a program point where a “free” operation is performed. The state A will transit to the state E (i.e., a memory leak issue) if a heap object goes out of its life scope and can be no longer referenced.
Given an FSM, we can build the sparse program representation, the use-flow graph, as follows.

**Definition 2** (Use-Flow Graph (UFG)). The use-flow graph of a program with regard to a given FSM is a directed graph $G = (V,E)$, where $V$ is the set of vertices and $E$ is the set of edges:

- A vertex $o \circ p \in V$ represents an object $o$ at a program point $p$. $p$ is one of the following program points:
  - $p \in P$ where $P \in \Sigma$ and $\delta(s,P) \neq s$. That is, $p$ is a program point that can cause a transition between different states.
  - $p$ is where we call a function using $o$ as an actual parameter.
  - $p$ is the entry of a function where $o$ is a formal parameter.
  - For each $o \circ p' \in V$, $p$ is the dominance frontier [21] of $p'$. Intuitively, the dominance frontier is where the states of the object $o$ from different paths can be merged.
- A directed edge $(o_1 \circ p_1, o_2 \circ p_2) \in E$ if and only if $o_1$ and $o_2$ represent the same object and there exists a control flow path from $p_1$ to $p_2$. The path does not go through any other program point $p_3$ such that there exists $o_3$ representing the same object and $o_3 \circ p_3 \in V$.

Given a program and the FSM of a property, the UFG can be built efficiently according to Definition 2 in polynomial time. The basic idea is to traverse the control flow graph of each function and remove the statements at program points that are irrelevant according to the FSM.

**Example 2.** Figure 5 illustrates an example of the UFG w.r.t. the FSM in Figure 4. As a sparse representation, all irrelevant program statements such as the ones at Lines 4 and 5 are not modeled. We have the vertices $t@s_7$ and $p@s_{12}$ because they are the dominance frontiers of $t@s_6$ and $p@s_{11}$, respectively. At the program points $s_7$ and $s_{10}$, the states of the heap object from different program paths can be merged. We have the vertex $t@s_9$ because this is the end point of the scope of the heap object $t$.

Next, we describe how to detect memory leaks using UFG. The basic idea is to check the state of a heap object by traversing the UFG inter-procedurally. When traversing a UFG, we use a set to keep track of the states at each vertex. From the memory allocation site, the state $A$ is added to the set. The state set is propagated forward along the UFG edges. When visiting a vertex corresponding to a transition of the FSM, we transit a state accordingly. Otherwise, the states remain unchanged. At each merge point, we merge the state sets from different paths by the set union operation. Instead of giving a complex formal representation of the algorithm, we use the following example to illustrate the process.

**Example 3.** We use Table I to illustrate the steps of our state analysis on the UFG in Figure 5. Each row of the table shows the state set before and after a vertex. For example, in Steps 5 and 7, since the program points, $s_{12}$ and $s_7$, are dominance frontiers where two state sets from different paths meet, we use the set-union operation to merge the states. In Step 8, since the program point $s_9$ represents the end of the scope of the heap object $t$, the state $A$ transits to the state $E$ and the state $F$ transits to the state $X$. Since the state $E$ is obtained via an *out-of-scope* operation, we report a memory-leak candidate. This candidate will be verified path-sensitively as detailed later.
As illustrated by the above example, the UFG is traversed inter-procedurally. The context-sensitivity is achieved via the CFL-reachability method \[22\]. That is, we assign a string to each state during the graph traversal to check the validity of the context. When propagating a state along a call edge at a call site \(cs\), we append a left parenthesis \((cs)\) to the string. When propagating a state back to a call site \(cs\) along a return edge, we append a right parenthesis \(cs)\) to the string. A state propagation is valid only if the string has matched parentheses.

We create a function summary for each function in a demand-driven way. That is, when reaching a call site, we check if the calleee has a usable summary so that we do not need to reanalyze the callee. Otherwise, we create a summary after the calleee is analyzed. In our approach, a summary is a map between the input states of a parameter \(p\) and the states of \(p\) at the end of a function. Therefore, it is unnecessary to repeat the analysis of a function at different call sites if its summary has been created.

Our analysis can be efficiently implemented using the RHS algorithm \[22\]. Given an FSM \(M = (\Sigma, \mathcal{S}, \delta, s_0, F)\) and its corresponding UFG \(G = (V, E)\), the time complexity of the algorithm is \(O(|V|^2 |S|(|E|) + |\text{Calls}| |S|)\), where |\text{Calls}| is the number of call sites in the program to check.

### B. Staged-Path-Sensitive Verification

```
1. void foo(bool c) {
2.     int *t = malloc(...);
3.     if (c)
4.         free(t);
5.     else if (!c)
6.         free(t);
7. } 
```

Fig. 6. A false memory leak case whose path condition contains an apparent contradiction \(c \land \neg c\).

Some of the UFG paths generated by the aforementioned state analysis are infeasible, which may result in false positives as illustrated in Figure 6. To reduce the false positives, we introduce a path-sensitive verification step where we collect and solve path conditions of each memory-leak candidate path. Although the path-sensitive verification is usually expensive, our approach can be efficient because of two important observations. First, we observe that the state analysis only produces a handful of memory leak candidates that require a path-sensitive verification. This is because most of the heap memory spaces are managed safely in practice. Second, we observe that the constraint in a path condition usually has apparent contradictions such as \(a \land \neg a\), as shown in the example in Figure 6. The reason is that the programmers tend to use direct contradictions to ensure some required logical properties. Thus, we first adopt a linear-time solver to detect apparent contradictions and to filter obviously infeasible paths. For the remaining complex cases, we use a fully-featured SMT solver, such as Z3 \[20\], to check the path feasibility. The basic idea of the linear-time solver is similar to the one used in prior study \[18\], and it continuously collects the sets of positive and negative atomic constraints during the construction of a path condition. An atomic constraint is a first-order logic formula that does not contain any logic operator like \(\land, \lor, \neg\). For example, \(a < 2b\) and \(c\) are two atomic constraints in \(a < 2b \land c\). If a path condition has an atomic constraint \(a\) in both sets, this path condition must contain \(a \land \neg a\) and, thus, is unsatisfiable.

### IV. IMPLEMENTATION

We have implemented SMOKE on top of the LLVM framework \[23\], which takes an LLVM-Bitcode file as input, for detecting C/C++ memory leaks.

Similar to some earlier work \[8\], \[9\], \[24\], we have some unsound trade-offs to make the detector more practical. We assume a path in UFG is safe if this path flows to a global pointer or a container (e.g., std::list, std::vector). We do not report memory leaks if a path ends in a call to `exit()`. If a heap object is used as an argument of a library function (i.e., a function that is not defined in the bitcode) on a path, we conservatively treat it as an unknown value and stop searching this path. We manually modeled some common library functions, such as `memcpy` and `memset`, to improve the precision and the recall. Similar to FASTCHECK \[8\], we do not check the arithmetic operations on heap pointer, `free(p + y)` is simply treated as `free(p)`.

Figure 7 illustrates the overview of our tool SMOKE. SMOKE has four phases: Pre-Analyses, UFG Construction, State Analysis and Path-Sensitive Verification. Here we only discuss the details of the Pre-Analyses phase since other three phases have already been discussed in Section III.

In the pre-analyses phase, we conduct several analyses to compute the necessary information for later phases. We first use a flow-sensitive and context-sensitive pointer analysis similar to the one used in PINPOINT \[18\] to compute the data dependence. Control dependence is computed on demand, since not all functions in a program are related to memory leak detection. We construct the control dependence by computing the dominance frontier in the reverse graph of the control flow graph \[21\]. To construct the call graph, we use the must-alias results from the pointer analysis to resolve function pointers, and adopt a class hierarchy analysis \[25\] to resolve virtual function calls.

During the state analysis, we only consider the case when the memory allocation operation is successful. For instance, if a heap object is created in \(x = malloc()\), we do not consider the case when the test \(x == NULL\) is true. We use a light-weight data-flow analysis to identify and ignore UFG edges on which heap memories have not been successfully allocated.

### V. EVALUATION

We evaluate the precision, the recall and the scalability of SMOKE by comparing four well-known static analysis tools
that have memory leak detectors, from both academia and industry. We compare to both SABER [9] and PINPOINT [18], since they are the state-of-the-art SVFA based approaches with the precision of flow-sensitivity and path-sensitivity, respectively. We also choose CSA [1] and INFER [2], two prominent and mature open-source tools from the industry. We plan to evaluate other memory leak detection tools, such as SATURN [3] and CALYSTO [26]. However, they are either publicly unavailable or outdated for running in the environments we are able to set up. To demonstrate the usefulness of our approach, we also seek confirmations from original developers of the subjects we use.

For each baseline, we only enable the memory leak detector, since some tools may have other bug detectors. We set the timeout to 2 hours, and configure each tool with its default settings. All the experiments are performed on a moderate computer running Ubuntu-16.04 with an Intel Core i5-6500 quad-core processor, and 64GB physical memory.

A. Subjects for Evaluation

We evaluate SMOKE, SABER, PINPOINT, CSA and INFER using twenty-nine subjects, including the twelve benchmark programs from standard SPEC CINT2000 [27], a commonly used benchmark in the existing literature [9], and seventeen well tested open-source projects. Table II shows the basic information of the evaluation subjects. The size of these subjects ranges from a few thousand of lines of code to nearly eight million of lines of code. These projects are widely used and well tested before release. Some of them are regularly scanned by free or commercial static tools such as COVERAGE SAVE* and, thus, expected to have high code quality. We divide the subjects into two categories: medium-sized projects (including all SPEC CINT2000 programs and the projects with code size less than 1MLoC) and large-sized projects with code size larger than 1MLoC.

B. Scalability

To evaluate the scalability of each tool, we first check whether it can successfully analyze the selected subjects under our experimental environment (which is a typical desktop computer rather than a powerful cluster) and within the time budget (2 hours). As shown in Table II, SMOKE successfully analyzes all twenty-nine projects in forty minutes, while others fail to analyze some of the projects due to either crash (out-of-memory or segmentation faults) or timeout. For example, SABER, having the lowest success rate, fails on nine projects, including two medium-sized and seven large-sized projects. PINPOINT and CSA achieve the second-best success rate, but still fail on four projects, which are mostly large-sized projects.

We further compare the time cost of analysis of all the tools shown in Table II. On average, SMOKE achieves the highest efficiency and is significantly (>5.2X faster than SABER, >13.0X faster than PINPOINT, >12.4X faster than CSA, >22.8X faster than INFER) faster than other tools. SMOKE performs better than SABER and PINPOINT in all subjects. Compared to CSA and INFER, SMOKE also performs better on all the subjects, except one medium-sized project, 253.perlbmk.

Overall, the evaluation results show that, compared to other tools, SMOKE achieves the highest scalability and scales to millions of lines of code without requiring too much computation resource (using a desktop computer).

C. Precision

Following the common practice in the literature on memory leak detection [8], [9], we manually check each error report provided by all the tools, and then compare their false positive rate (FP rate for short) to evaluate the precision. This process may be subjective and introduce threats to the validity of the FP results. Therefore, we ask three software engineers to cross-validate the results and, meanwhile, seek confirmations from the developers of the subjects. In addition, we release the reports and data for inspection.†

*https://scan.coverity.com/projects/
†https://smokeml.github.io/data
Table II
ANALYSIS TIME COMPARISON

<table>
<thead>
<tr>
<th>SPEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT2000</td>
</tr>
<tr>
<td>164.gzp</td>
</tr>
<tr>
<td>175.vpr</td>
</tr>
<tr>
<td>176.gcc</td>
</tr>
<tr>
<td>181.mdf</td>
</tr>
<tr>
<td>186.crafty</td>
</tr>
<tr>
<td>197.parser</td>
</tr>
<tr>
<td>252.eon</td>
</tr>
<tr>
<td>253.perlbbm</td>
</tr>
<tr>
<td>254-gap</td>
</tr>
<tr>
<td>255.vortex</td>
</tr>
<tr>
<td>256.bzip2</td>
</tr>
<tr>
<td>300.twolf</td>
</tr>
<tr>
<td>Bftpdk</td>
</tr>
<tr>
<td>Htop</td>
</tr>
<tr>
<td>Memcached</td>
</tr>
<tr>
<td>Caffe</td>
</tr>
<tr>
<td>LAME</td>
</tr>
<tr>
<td>Zlib</td>
</tr>
<tr>
<td>Tmux</td>
</tr>
<tr>
<td>Apache httpd</td>
</tr>
<tr>
<td>OpenSSL</td>
</tr>
</tbody>
</table>

Table III shows the comparison results. On average, SMOKE achieves an FP rate of 24.4%, which is the lowest one among all the tools. As discussed in Section V-B, not all the subjects can be successfully analyzed by all the tools. And the number of false positives in the failure cases (e.g., SABER fails in the project LAME) is counted as 0. For the medium-sized subject, the small number of failure cases (only 2 by SABER, 1 by CSA and INFER) allows a fair comparison of all the tools, where SMOKE outperforms all other tools with a lowest FP rate of 15.0%. The second best tool, PINPOINT, achieves a much higher FP rate of 41.7%. INFER performs the worst, with an FP rate of 87.5%. For the large-sized subjects, we also observe SMOKE achieves the lowest FP rate, which is consistent with the results in medium-sized subjects.

We further investigate why SMOKE can achieve the lowest FP rate. SABER does not capture the path correlation. Therefore, it cannot filter out cases with infeasible paths. Since PINPOINT and CSA achieve the precision of path-sensitivity, they report the smallest number of false positives. However, they report much fewer true positive instances, incurring the higher FP rate. INFER has the highest FP rate. A manual check reveals that INFER generates a specification for each analyzed function, and discards some of these specifications for unknown reasons. The omitted specifications produce imprecision in the whole program analysis. For example, a typical false positive case happens when the specification for a wrapper function of free (i.e., a library function to free heap object) is discarded.

D. Recall

From the results in Table III, we observe that SMOKE reports more memory leaks than other tools. SMOKE reports 158 real memory leaks while the other four tools report only 48 real memory leaks in total. SABER and PINPOINT make significant precision trade-offs to achieve efficiency. For instance, they limit the calling depth for inter-procedural analysis and only report memory leaks with six levels of function calls. They also stop analysis whenever they find that a heap object is assigned to a global variable on a path, even when the heap object obviously leaks on another path. For example, Figure 8 shows a leak found by SMOKE: A heap object allocated in get.dll name flows to dll name. It leaks when taking the true branch at line 207.
TABLE III
MEMORY LEAK RESULTS

<table>
<thead>
<tr>
<th>Origin</th>
<th>Project</th>
<th>SMOKE</th>
<th>Saber</th>
<th>PINPOINT</th>
<th>CSA</th>
<th>Infer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>%FP</td>
<td>%FP</td>
<td>%FP</td>
<td>%FP</td>
<td>%FP</td>
</tr>
<tr>
<td>SPEC CINT2000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>164.gzip</td>
<td>1/1</td>
<td>1/2</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
</tr>
<tr>
<td>175.vpr</td>
<td>0/2</td>
<td>0/0</td>
<td>0/0</td>
<td>1/1</td>
<td>0/0</td>
<td></td>
</tr>
<tr>
<td>176.gcc</td>
<td>0/8</td>
<td>3/7</td>
<td>0/3</td>
<td>0/1</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>181.mcf</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td></td>
</tr>
<tr>
<td>186.crafty</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td></td>
</tr>
<tr>
<td>197.parser</td>
<td>1/4</td>
<td>0/1</td>
<td>2/3</td>
<td>0/0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>252.eon</td>
<td>0/1</td>
<td>0/1</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
</tr>
<tr>
<td>253.parl/ikm</td>
<td>0/1</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
</tr>
<tr>
<td>254.gap</td>
<td>0/0</td>
<td>1/1</td>
<td>0/0</td>
<td>0/0</td>
<td>1/1</td>
<td></td>
</tr>
<tr>
<td>255.vortex</td>
<td>0/1</td>
<td>0/2</td>
<td>1/1</td>
<td>0/0</td>
<td>0/0</td>
<td></td>
</tr>
<tr>
<td>256.bzip2</td>
<td>0/1</td>
<td>0/1</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td></td>
</tr>
<tr>
<td>300.xcom</td>
<td>0/0</td>
<td>1/3</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td></td>
</tr>
<tr>
<td>Bftpd (&lt;1 LoC)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memcached</td>
<td>1/2</td>
<td>3/4</td>
<td>0/1</td>
<td>0/0</td>
<td>2/3</td>
<td></td>
</tr>
<tr>
<td>Caffe</td>
<td>0/0</td>
<td>0/0</td>
<td>1/1</td>
<td>N/A</td>
<td>0/0</td>
<td></td>
</tr>
<tr>
<td>LAME</td>
<td>0/0</td>
<td>N/A</td>
<td>0/0</td>
<td>0/0</td>
<td>1/1</td>
<td></td>
</tr>
<tr>
<td>Zlib</td>
<td>0/0</td>
<td>1/1</td>
<td>0/0</td>
<td>0/0</td>
<td>1/2</td>
<td></td>
</tr>
<tr>
<td>tmux</td>
<td>2/9</td>
<td>6/6</td>
<td>2/2</td>
<td>0/0</td>
<td>17/18</td>
<td></td>
</tr>
<tr>
<td>Apache httpd</td>
<td>0/1</td>
<td>4/5</td>
<td>0/1</td>
<td>1/2</td>
<td>5/6</td>
<td></td>
</tr>
<tr>
<td>OpenSSL</td>
<td>2/6</td>
<td>N/A</td>
<td>1/3</td>
<td>0/0</td>
<td>9/10</td>
<td></td>
</tr>
<tr>
<td>Total (&lt;1 LoC)</td>
<td>6/40</td>
<td>23/35</td>
<td>5/12</td>
<td>4/8</td>
<td>42/48</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Location: import.c:277</th>
</tr>
</thead>
<tbody>
<tr>
<td>195. /* add a dll to the list of imports */</td>
</tr>
<tr>
<td>196. void add_import_dll( const char *name, const char *filename )</td>
</tr>
<tr>
<td>197. {</td>
</tr>
<tr>
<td>198. DSLSPEC *spec;</td>
</tr>
<tr>
<td>199. char *dll_name = get_dll_name( name, filename );</td>
</tr>
<tr>
<td>200. struct import *imp = malloc(sizeof(*imp));</td>
</tr>
<tr>
<td>201. int set_imp = xmalloc(sizeof(*set_imp));</td>
</tr>
<tr>
<td>202. memset( imp, 0, sizeof(*imp) );</td>
</tr>
<tr>
<td>203.</td>
</tr>
<tr>
<td>204. if (dll_name)</td>
</tr>
<tr>
<td>205. set_imp-&gt;full_name = strdup( filename );</td>
</tr>
<tr>
<td>206.</td>
</tr>
<tr>
<td>207. elsif (dll_name)</td>
</tr>
<tr>
<td>208.</td>
</tr>
<tr>
<td>209. free(imp);</td>
</tr>
<tr>
<td>210. return;</td>
</tr>
<tr>
<td>211. )</td>
</tr>
<tr>
<td>212.</td>
</tr>
<tr>
<td>213. imp-&gt;dll_name = spec-&gt;file_name ? spec-&gt;file_name : dll_name;</td>
</tr>
<tr>
<td>214. imp-&gt;c_name = make_c_identifier( imp-&gt;dll_name );</td>
</tr>
<tr>
<td>215.</td>
</tr>
<tr>
<td>216. if (is_delayed_import( dll_name ))</td>
</tr>
<tr>
<td>217. list_add_tail( &amp;dll_delayed, &amp;imp-&gt;entry );</td>
</tr>
<tr>
<td>218. else</td>
</tr>
<tr>
<td>219. list_add_tail( &amp;dll_imports, &amp;imp-&gt;entry );</td>
</tr>
<tr>
<td>220. )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Location: commands.c:71</th>
</tr>
</thead>
<tbody>
<tr>
<td>69. char *bftpd_cwd_mappath(char *path)</td>
</tr>
<tr>
<td>70. {</td>
</tr>
<tr>
<td>71. char *result = malloc(strlen(path));</td>
</tr>
<tr>
<td>72. char *path2;</td>
</tr>
<tr>
<td>73. char *tmp;</td>
</tr>
<tr>
<td>74. if (path)</td>
</tr>
<tr>
<td>75. path2 = strdup(path);</td>
</tr>
<tr>
<td>76. if (path2)</td>
</tr>
<tr>
<td>77. return;</td>
</tr>
<tr>
<td>78. return NULL;</td>
</tr>
<tr>
<td>79. appendpath(result, path2);</td>
</tr>
<tr>
<td>80. free(path2);</td>
</tr>
<tr>
<td>81. return result;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Location: commands.c:1522</th>
</tr>
</thead>
<tbody>
<tr>
<td>1516. void command_rnto(char *newname)</td>
</tr>
<tr>
<td>1517. {</td>
</tr>
<tr>
<td>1518. char mapped = bftpd_cwd_mappath(newname);</td>
</tr>
<tr>
<td>1519. if (mapped)</td>
</tr>
<tr>
<td>1520. control_printf(SL_FAILURE, &quot;Error: Unable to rename file.&quot;);</td>
</tr>
<tr>
<td>1521. return NULL;</td>
</tr>
<tr>
<td>1522. }</td>
</tr>
</tbody>
</table>

SABER and PINPOINT cannot find this leak since it can flow to a global variable (dll_delayed or dll_imports) when taking the false branch at line 207. CSA does not analyze function calls across different files. INFER can only handle a small number of function calls across different files. However, we observe that it is very common that a memory leak relates to functions from different files. As a result, CSA and INFER miss many real memory leaks. For example, Figure 9 shows a memory leak in Bftpd reported by SMOKE but not by CSA or INFER. A heap object allocated in a source file cmd.c leaks in another file commands.c.

E. Contribution of the Analysis Stages

To better understand the effectiveness and the efficiency of the staged analysis, we report the number of pruned candidates and the time cost of each stage in Table IV. Note that, since it is difficult to calculate the total number of the candidate paths starting from each allocation sites (e.g., path explosion due to loops), we estimate the lower bound of the total number of candidates using the number of allocation sites.
TABLE IV
STATISTICS OF TWO ANALYSIS STAGES

<table>
<thead>
<tr>
<th>Project</th>
<th># Estimated Paths</th>
<th>1st Stage</th>
<th>2nd Stage</th>
<th>2nd Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td># Paths</td>
<td>Time (sec)</td>
<td>% Total Time</td>
</tr>
<tr>
<td>164.gzip</td>
<td>14</td>
<td>5</td>
<td>0.34</td>
<td>41.8%</td>
</tr>
<tr>
<td>175.vp8</td>
<td>3</td>
<td>2</td>
<td>2.73</td>
<td>97.5%</td>
</tr>
<tr>
<td>176.vdc</td>
<td>96</td>
<td>13</td>
<td>32.50</td>
<td>98.0%</td>
</tr>
<tr>
<td>181.mcf</td>
<td>4</td>
<td>0</td>
<td>0.07</td>
<td>100.0%</td>
</tr>
<tr>
<td>186.crafty</td>
<td>0</td>
<td>0</td>
<td>0.11</td>
<td>100.0%</td>
</tr>
<tr>
<td>197.parser</td>
<td>148</td>
<td>3</td>
<td>1.82</td>
<td>97.6%</td>
</tr>
<tr>
<td>252.dion</td>
<td>142</td>
<td>55</td>
<td>6.23</td>
<td>75.9%</td>
</tr>
<tr>
<td>253.bergmk</td>
<td>17</td>
<td>1</td>
<td>117.66</td>
<td>100.0%</td>
</tr>
<tr>
<td>254.gap</td>
<td>1</td>
<td>0</td>
<td>0.71</td>
<td>100.0%</td>
</tr>
<tr>
<td>255.vortex</td>
<td>11</td>
<td>5</td>
<td>0.97</td>
<td>85.1%</td>
</tr>
<tr>
<td>256.bsp2</td>
<td>11</td>
<td>1</td>
<td>0.09</td>
<td>96.2%</td>
</tr>
<tr>
<td>300.bf25</td>
<td>162</td>
<td>0</td>
<td>0.35</td>
<td>100.0%</td>
</tr>
<tr>
<td>Bfdp</td>
<td>87</td>
<td>1</td>
<td>0.66</td>
<td>97.7%</td>
</tr>
<tr>
<td>Htop</td>
<td>12</td>
<td>1</td>
<td>4.59</td>
<td>99.6%</td>
</tr>
<tr>
<td>Memcached</td>
<td>59</td>
<td>3</td>
<td>2.50</td>
<td>93.7%</td>
</tr>
<tr>
<td>Caffe</td>
<td>910</td>
<td>0</td>
<td>71.24</td>
<td>100.0%</td>
</tr>
<tr>
<td>LAME</td>
<td>34</td>
<td>0</td>
<td>13.99</td>
<td>100.0%</td>
</tr>
<tr>
<td>Zlib</td>
<td>1</td>
<td>0</td>
<td>0.03</td>
<td>100.0%</td>
</tr>
<tr>
<td>Tmux</td>
<td>108</td>
<td>9</td>
<td>16.53</td>
<td>84.7%</td>
</tr>
<tr>
<td>Apache httpd</td>
<td>10</td>
<td>1</td>
<td>11.46</td>
<td>100.0%</td>
</tr>
<tr>
<td>OpenSSL</td>
<td>24</td>
<td>10</td>
<td>252.79</td>
<td>96.5%</td>
</tr>
<tr>
<td>FFmpeg</td>
<td>3</td>
<td>3</td>
<td>170.61</td>
<td>98.0%</td>
</tr>
<tr>
<td>Godot Engine</td>
<td>666</td>
<td>24</td>
<td>1690.00</td>
<td>99.5%</td>
</tr>
<tr>
<td>MySQL</td>
<td>269</td>
<td>61</td>
<td>273.78</td>
<td>97.4%</td>
</tr>
<tr>
<td>Chrome V8</td>
<td>6,152</td>
<td>32</td>
<td>2533.49</td>
<td>98.0%</td>
</tr>
<tr>
<td>Skia</td>
<td>6,670</td>
<td>43</td>
<td>354.19</td>
<td>89.9%</td>
</tr>
<tr>
<td>Blender</td>
<td>4,638</td>
<td>16</td>
<td>2032.35</td>
<td>99.9%</td>
</tr>
<tr>
<td>Wine</td>
<td>2,011</td>
<td>133</td>
<td>259.44</td>
<td>98.3%</td>
</tr>
<tr>
<td>Firefox</td>
<td>23,019</td>
<td>188</td>
<td>2377.78</td>
<td>99.2%</td>
</tr>
</tbody>
</table>

| Total       | 43,282            | 610       | 352.7      | 94.6%       | 5           | 2.4%       |

Average: 43.282 610 352.7 94.6% 5.2 5.4%

of the stages are effective in improving the precision.

Table IV shows the time cost of the two stages. SMOKE spends 94.6% of the time in the first phase and only 5.4% in the second phase. The two-staged design significantly reduces the cost for the path-sensitive analysis and, thus, achieves high efficiency.

F. Detected Real MemoryLeaks

To better understand the usefulness of SMOKE in practice, we seek confirmations from the original developers of the subjects. Since we can not flood them with all warnings, we manually pre-screen the bug reports and choose only the ones that are likely to have severe impacts. The majority of the reports are for large and well-maintained projects such as FFmpeg, Wine, Firefox, MySQL, Godot Engine and Chrome V8. All thirty reports get confirmed by their developers and result in many patches and bug fixes. This result confirms the usefulness of our approach because those projects are regularly scanned by free and commercial tools, and SMOKE only takes 3 ~ 40 minutes to analyze each project. We release the confirmed memory leaks online.

Due to the high scalability of SMOKE, we can detect the inter-procedural memory leaks in large projects without consuming too much computing resources. Figure 10 shows a memory leak confirmed by the developers of Firefox, a project of approximately eight million lines of code. In this example, the memory space pointed-to by videoData is allocated at Line 279 and leaks on an early return at Line 286 when timestamp.isValid() returns false. This case is not very complicated. However, because of the complexity in the code and the enormous project size, this memory leak has been hidden for more than one year. Figure 9 shows a memory leak that SMOKE found in Bftpd, a lightweight, flexible FTP server widely used in desktops, servers, embedded devices, and media centers. This vulnerability has been in the code since 2005. It has survived for more than twelve years and has impacted eighty-six versions altogether. We report this leak to its developers and receive a confirmation and an appreciation acknowledgment on its homepage. A remote attacker can utilize this leak to launch denial-of-service attacks to the users of Bftpd. Due to its high severity, a CVE ID (CVE-2017-16892) is assigned.

---

Footnote:

1. https://smokeml.github.io/list

---

Fig. 10. A memory leak in Firefox

Fig. 11. A false positive report in Blender
G. Limitations

Our approach can be imprecise for several reasons. One reason is the lack of library specifications. Figure 11 shows a leak-free example simplified from a false positive report in Blender. At Line 431, the program allocates an array and stores it to the memory pointed-to by outerIndexPtr, under the condition that isCompressed() returns true. At Line 440, the program “deletes” outerIndexPtr when another call to isCompressed() returns true at Line 439. The function isCompressed() is implemented in a library, and it returns the same boolean value at both places. Hence, the two branches are correlated. Unfortunately, we cannot statically determine this branch correlation since we do not have the specification of the library function. One way to mitigate this problem is to manually or automatically provide specifications for libraries when detecting infeasible paths.

Another reason is that we inherit the imprecision from the lower-level program analyses such as the pointer analysis. The pointer analysis may mistakenly identify two different heap objects as aliases. Operations on one object of the alias pair may cause state changes of another, which could result in mutual interference and incorrect results.

There are still some types of infeasible paths that cannot be identified by our approach. Such paths usually involve complex arithmetic in branch conditions, complicate data dependence or deep inter-procedural effects. Including more control and data dependence in the path constraints can help to mitigate this limitation, at the price of being less efficient.

VI. Related Work

In this section, we survey the related approaches in two categories.

A. Static Memory Leak Detection

SATURN [3] reduces the memory leak detection problem to a boolean satisfiability problem and uses a SAT-solver to find memory leaks. Facebook Infer [2] is based on abductive inference of separation logic that extends the Hoare Logic by explicitly modeling the heap. CLAUSEAU [4], [5] detects memory leaks based on a practical ownership model of memory management. Orlovich and Rugina [6] proposed a leak detection algorithm based on a reverse data-flow analysis that assumes the presence of the leak first, followed by a pruning method. SPARROW [7] detects memory leaks in C programs based on abstract interpretation. Tools based on symbolic execution such as the CSA [1], COVERAGE SAVE [28], and KLEE [29] are promising in practice. By treating all external inputs as symbols and execute the program on symbols, symbolic execution tools can explore program states that are hard to reach for concrete executions. However, symbolic execution tools do not scale to large programs due to the path explosion problem and intensive uses of the constraint solver. All of the above techniques are not sparse, which analyze a lot of irrelevant program statements and, thus, suffer from performance issues.

FASTCHECK [8], SABER [9], and PINPOINT [18] are mostly related to our approach. They work on a sparse value flow graph with guards annotated on the graph edges. As we have discussed in this paper, when high precision is required, VFG-based model of the memory leak problem [8], [9] is the reason for the analysis to be not scalable. To achieve scalability, FASTCHECK and SABER have to compromise the path-sensitivity, leading to many false warnings. PINPOINT improves the sparse value-flow analysis by building precise data dependence in an efficient manner, but it does not provide a better model for memory leak detection. In comparison, we propose a FSM-based model that enables us to efficiently detect memory leaks on UFG.

B. Dynamic Memory Leak Detection

Dynamic approaches [24], [30]–[36] detect memory leaks by instrumenting and running a program. Some dynamic approaches operate at the binary level, such as the memcheck tool of VALGRIND [31], DR. MEMORY [30] and PURIFY [33]. These approaches track memory allocation and deallocation during a program’s execution, and detect leaks by scanning the program’s heap for memory blocks that do not pointer points to. INSURE++ [36] and ADDRESS SANITIZER [35] detect memory leaks by inserting extra statements to the source code before compiling the binary.

Unlike static approaches that have relatively high false positive rate due to the abstraction of concrete program states, dynamic approaches have few false positives because they have access to the concrete program states at runtime. However, dynamic approaches can miss many real bugs because they cannot cover all possible program behaviors with limited number of test cases. Also, dynamic approaches are hard to be applied in a production run, because running the instrumented program usually causes unbearable runtime overhead.

VII. Conclusion

We have presented an approach to static memory leak detection, which runs in a fast, scalable, and precise manner. The key factor to make our technique fast is a staged analysis, in which we first efficiently filter safe cases based on our new program representation, i.e., use-flow graph, and then employ a constraint solver to verify path feasibility only for a handful of leak candidates. We implemented our technique as a tool, called SMOKE, and evaluated it systematically. The evaluation results demonstrate that SMOKE is promising as an industrial-strength static memory-leak detector.

Acknowledgment

We thank the anonymous reviewers for their insightful comments and Dr. Yulei Sui for his help on the SABER implementation. This work was partially funded by Hong Kong GRF16214515, GRF16230716, GRF16206517, NSFC61628205 and ITS/215/16FP grants.